Programação de Sistemas – 24/25
Project report

Diogo Ramos (100299) & Francisco Tavares (103402) – Group: 1

1 Introduction

This project is a two-part assignment where we develop a simple version of the classic Space Invaders game. Players control astronauts who try to shoot randomly moving aliens, earning points for each successful hit. The game ends when all astronauts are defeated, and the player with the highest score wins.

In **Part A**, we create the basic game. A server manages the game logic, and clients let players control astronauts and view the game.

In **Part B**, we improve the game by adding threads for better performance and responsiveness. We also add a new high-scores feature to store and display top players while keeping the Part A clients compatible with the updated server.

This project helps us learn about systems programming, threads, and building networked applications.

2 Implemented functionalities

2.1 Part A

Table 1: Implemented functionalities (PART A)

	Not implemented	With faults	Totally Correct
game-server.c			
Suitable sockets			X
Assignment of Astronaut letters			X
Storage of clients Astronauts and Aliens			X
Management of astronauts			X
astronaut-client.c			
Connect			X
Movement			X
Zap			X
Disconnect			X
Display score			X
outer-space-display.c + display on game-server.c			
Correct update of screen			X
Zapps			X
Destruction of aliens			X
Rules			
Movement of Aliens			X
Astronaut zapping			X
Astronaut zapping (fire rate)			X
Astronaut zapping (0.5 second ray)			X
Aliens are destruction (points)			X
Astronauts stunning			X
Misc			
Messages validation			X
Optimization of communication			X
Code organization			X
Functions return values validation			X

2.2 Part B

Table 2: Implemented functionalities (PART B)

	Not implemented	**With faults**	**Totally Correct**
game-server.c			
Suitable sockets			X
Disconnect of clients			X
Communication thread			X
Aliens thread			X
Synchronization			X
astronaut-display-client.c			
Connect			X
Movement			X
Zap			X
Disconnect			X
Update of display			X
Display score			X
Threads			X
space-high-scores			
Client on other language			X
Protocolbuffer messages			X
Rules			
Movement of Aliens			X
Astronaut zapping			X
Astronaut zapping (fire rate)			X
Astronaut zapping (0.5 second ray)			X
Aliens destruction and points			X
Astronauts stunning			X
Aliens population recovery			X
Misc			
Messages validation			X
Optimization of communication			X
Code organization			X
Functions return values validation			X

2.3 Description of faulty functionalities

We have no faulty functionalities.

3 PART A Description of code

3.1 Data types

In this section students should present a list and describe the various datatype (structs) and lists/array used to store the Astronauts, aliens and clients

Astronauts and Aliens

Both astronauts and aliens are stored in separate arrays of type ch_info_t:

- **Astronauts**: Stored in an array of size MAX_PLAYERS.
- **Aliens**: Stored in an array of size ALIEN_COUNT.

The ch_info_t structure holds the following information:

- **ch**: Unique identifier for the astronaut/alien.
- **pos_x, pos_y**: Current position (x, y) of the astronaut/alien in the game.
- **dir**: Indicates whether the astronaut/alien moves vertically (1) or horizontally (0).
- **score**: The astronaut's current score. *(Not applicable for aliens)*
- **last_fire_time**: Timestamp of the astronaut's last shooting action. *(Not applicable for aliens)*
- **stunned**: Timestamp indicating when the astronaut was stunned (if applicable). *(Not applicable for aliens)*
- **GAME_TOKEN**:
    - For astronauts: Authenticates the astronaut's connection.
    - For aliens: Authenticates that the client controlling alien movements is the child process of the server program.

Clients

Clients are identified by the corresponding ch variable of the astronaut they control. Each client's messages are authenticated using a GAME_TOKEN provided when the client connects to the server. This token remains constant throughout the game and ensures that:

- Only authorized clients can control the assigned astronaut.
- Unauthorized actions, such as attempting to control another astronaut, are prevented.

3.2 Functions List

In this section students should present a list with every implemented function, divided by the various components implemented:

- game-server.c
- astronaut-client.c
- outer-space-display.c.

For each function students should describe the objective of each function.

game-server.c

main: Sets up the server environment, initializes ZeroMQ sockets, loads game data, and enters the main loop to handle incoming requests.

remove_astronaut: Removes an astronaut from the game. Updates the game state and the display accordingly.

update_scoreboard: Updates the scoreboard with the current scores of all players.

fire_laser: Handles the firing of a laser by an astronaut, updating the game state and alien positions.

find_ch_info: Finds the index of a specific character in the char_data array.

new_position: Calculates the new position of a character based on the movement direction.

---

astronaut-client.c

main: Sets up the client environment (ZeroMQ, ncurses), sends initial requests to the server, and loops to process user input and server replies.

initialize_zmq: Initializes the ZeroMQ context and requester socket, connecting to the server.

initialize_ncurses: Sets up the ncurses environment for terminal-based user interaction.

cleanup: Cleans up ZeroMQ variables and releases ncurses resources.

handle_input: Processes keyboard input to create movement or action commands.

process_server_messages: Sends messages to the server and processes the server's replies.

---

outer-space-display.c

main: Initializes the display environment, subscribes to server updates using ZeroMQ, and draws game updates (including scores) in an ncurses window.

3.3 Implementation details

Description of the implementation of the astronaut zapping fire rate.

Description of the implementation of the astronaut zapping 0.5 second ray display

Description of the implementation of the astronauts stunning (immobility)

Description of the Implementation of the Astronaut Zapping Fire Rate

The astronaut's zapping is controlled by a fire rate mechanism. The fire rate is implemented by tracking the last_fire_time of each astronaut. When a player sends a firing request (MSG_TYPE_ZAP), the server compares the current time (current_time) with the last_fire_time. If the difference exceeds 3 seconds, the astronaut is allowed to fire. This ensures that each astronaut can only fire once every 3 seconds.

Description of the Implementation of the Astronaut Zapping 0.5-Second Ray Display

The fire_laser function handles this behaviour. Depending on the astronaut's movement direction:

- If the astronaut moves horizontally, the laser is displayed vertically as |.
- If the astronaut moves vertically, the laser is displayed horizontally as -.

The laser remains visible for 0.5 seconds using the usleep(500000) function and is then cleared from the screen.

Description of the Implementation of Astronaut Stunning (Immobility)

When an astronaut's laser hits another astronaut, the hit astronaut is stunned for 10 seconds. The stunning mechanism is implemented by updating the stunned field in the ch_info_t structure with the current timestamp when the hit occurs. The server ensures that stunned astronauts cannot perform any actions until the stun duration has elapsed by checking the difference between the current time and the stunned timestamp.

4 PART B Description of code

4.1 Data types

In this section students should present a list and describe the various datatype (structs) and lists/array that were added or modified in server of the Part B of the project.

The only modification done in Part B in this matter was adding a shared_data_t struct that was used by the threads in the server.

This structure serves as a shared resource for managing game state across threads. It includes:

- ch_info_t aliens[256]; // Array of alien data like in Part A (maximum of 256 aliens)
- int alien_count; // Current number of aliens alive
- pthread_mutex_t lock; // Mutex to synchronize access to shared resources
- int grid[16][16]; // 2D grid representing the game area for the aliens
- WINDOW *my_win; // Pointer to the main game window
- WINDOW *score_win; // Pointer to the score display window
- void *publisher; // Publisher for screen updates (for the astronaut and display clients)
- void *publisher2; // Secondary publisher (for the space-high-scores program)
- time_t last_alien_kill; // Timestamp of the last alien killed

This structure provides thread-safe access to critical game state variables. The aliens array stores information about all aliens currently in the game. The alien_count tracks the number of active aliens. The grid represents the game area that the aliens can occupy as a 16x16 2D array. Each cell indicates whether it is occupied and by an alien. The my_win and score_win manage the display of the game area and the scoreboard. The lock ensures that modifications to shared resources (e.g., alien data or grid state) are performed without race conditions. The publisher and publisher2 facilitate broadcasting updates (e.g., screen changes, scores) to connected clients.

4.2 Functions List

In this section students should present a list with new of modified functions from Part A to Part B:

- game-server.c
- astronaut-display-client.c
- space-high-scores.c

For each function students should describe the objective of each function.

game-server.c

main: Initializes the server environment. Sets up ZeroMQ publishers, creates shared data, and starts the message and alien movement threads.

message_thread: Handles incoming messages from clients. It sets up the game environment, processes user actions (movement, disconnection), and updates the game state accordingly.

alien_movement_thread: Randomly moves aliens around the grid. It locks shared data, updates alien positions, redraws them on the screen, and spawns additional aliens.

---

astronaut-display-client.c

main: Sets up ZeroMQ connections, receives character info from the server, initializes ncurses, runs an input loop, and spawns a display thread for incoming updates.

initialize_zmq: Creates the ZeroMQ context and requester socket, then connects to the server.

initialize_ncurses: Sets up the ncurses environment (window mode, no echo, etc.).

cleanup: Cleans up the ncurses environment and ZeroMQ resources (sockets, context).

handle_input: Reads keyboard input, updates the message with movement or zap commands, or signals disconnection.

process_server_messages: Sends the current message to the server and processes its numeric reply (score or disconnection).

display_thread: Subscribes to server updates, creates windows for the game and score displays, and refreshes them based on received updates.

---

space-high-scores.c

main: Subscribes to score updates from the server, parses the messages, and displays the latest scores in an ncurses window.

4.3 Implementation details

Description of the implementation of the astronaut zapping fire rate.

Description of the implementation of the astronaut zapping 0.5 second ray display.

Description of the implementation of the astronauts stunning (immobility).

Description of the implementation of the aliens destruction and points

Description of the implementation of the aliens population recovery

Description of the Implementation of the Astronaut Zapping Fire Rate

The astronaut's fire rate is controlled by the time between laser shots. When a player requests to fire a laser (MSG_TYPE_ZAP), the system checks the time since their last fire using difftime(current_time, char_data[ch_pos].last_fire_time). If the time difference exceeds 3 seconds, the laser is fired, and the last_fire_time is updated to the current time. This ensures that the astronaut can only fire one laser every 3 seconds.

Description of the Implementation of the Astronaut Zapping 0.5-Second Ray Display

When the astronaut fires the laser, the ray is shown for 0.5 seconds using usleep(500000) after it is rendered on the screen. The laser is displayed as either a vertical (|) or horizontal (-) line depending on the astronaut's direction. After 0.5 seconds, the laser is cleared from the screen.

Description of the Implementation of the Astronaut's Stunning (Immobility)

When an astronaut is hit by another astronaut's laser, they are stunned, i.e., made immobile. The code checks if any other astronaut occupies the same position as the firing astronaut and, if so, sets the stunned field of the affected astronaut to the current time (time(NULL)).

Description of the Implementation of the Aliens' Destruction and Points

When the astronaut's laser collides with an alien, the alien is removed from the game, and the corresponding grid cell is marked as empty. The alien count decreases, and the astronaut's score is incremented. Additionally, the time of the last alien kill is recorded, which is used to control alien population recovery (described below).

Description of the Implementation of the Aliens' Population Recovery

If no aliens are killed within 10 seconds (checked using difftime(now, data->last_alien_kill) >= 10), the alien population is replenished by approximately 10%. New aliens are spawned at random grid positions, and the grid is updated accordingly. The alien count is increased, and the aliens are displayed on the screen.

4.4 Threads

In the section students should present a list of every thread implemented, divided by components:

- astronaut-display-client.c
- game-server.c

Students should describe its overall functioning and objectives of each thread.

Threads in austronaut-display-client.c

There are 2 threads in this program.

The main() thread is responsible for handling input from the keyboard and dealing with the replies from the server.

The display_thread() is responsible for show the game screen via the messages received in the subscriber socket, and updating the ncurses windows.

Threads in game-server.c

This program has 4 threads.

The main() thread is only responsible for initializing the publisher sockets and calling the other 3 threads and waiting for them to finish.

The message_thread executes most of the important functions. Firstly, it initializes the windows and sockets. It receives the messages in the responder socket and handles the various types of messages received, calling the necessary functions for each one. At the end of each iteration it updates the score window and the display window.

The alien_movement_thread() is responsible for, in intervals of 1 second, moving all the aliens and updating the alien grid. It also sends the alien movement updates to the clients. Lastly, it also checks if an alien was killed in the last 10 seconds and repopulates if that is not the case.

The exit_thread() is responsible for handling the keyboard input in the server application and, if ‘q’ is pressed it closes all the other threads and sends messages to the other apps to let them know the game has ended, and they should terminate.

4.5 Shared variables

In this section students should present all shared variables that are accessed by multiple threads, divided by components.
Students should describe the objective of each variable and what threads access it.

Shared variables in austronaut-display-client.c

The only shared variable in this program in the context so the display thread can create the subscriber socket,

Shared variables in game-server.c

In this program we use a variable of type shared_data_t which is a structure. It contains all the alien information so it can be used by message_thread and alien_movement_thread. It contains alien positioning(aliens), the number of aliens(alien_count), the grid which represents if there is is an alien in each position(grid), and the last time an alien died (last_alien_kill). It also contains the pointers to the context and publisher contexts as multiple threads will publish. It contains the 2 ncurses windows and also a mutex to control variable access.

4.6 Synchronization

In this section students should present all synchronization variables used: mutextes.

Students should describe what type of guarding of synchronization they implement, what variables are related and on what functions they are accessed.

Synchronization in astronaut-display-client.c and game-server.c

Both the mutex in these programs are used to control access to the ncurses windows to avoid overlapping updates of those windows. This avoids bugs in the screen display.

We also limit access to the ‘grid’ and ‘aliens’ shared variables which contain the alien information and are accessed by both the alien_movement_thread and the fire_laser function which is called by a different thread. We need this as these variables can be accessed in parallel.

5 PART A Communication

5.1 Sockets

In this section student should decribe the sockets used in Part A.

Sockets utilized in part A

The server has a responder socket to receive and reply to the connection messages sent by astronaut-client and outer-space-display. It also receives, and replies to the zap, move and disconnect messages. The server also has a publisher socket to alert outer-space-display.c of the screen updates, be that movement zaps or alien movements. This socket is also resonible for sending the game over message.

The astronaut-client.c only has a requester socket to connect to the server and send the movement and zap messages, it also receives the score updates and replies formn the server.

The outer-space-display has a subscriber socket to subscribe to all topics and receive the screen and score updates from the server as well as the game over message.

5.2 Transferred data

In this section students should present the exchanges data structures/messages and relate them to the sockets described earlier.
 Students can present the typedef used and relate them to the messages presented in the assignments.

Transferred data in the requester/responder sockets.

The clients always send an object of type remote_char_t :

```
typedef struct

{

   msg_type_t msg_type;

   int ch;

   direction_t direction;

   int GAME_TOKEN;

} remote_char_t;
```

The direction is only utilized in case of movement measures, ch contains the character being controlled. GAME_TOKEN, is the token for authentication.

To the connection message the server replies with an object of type ch_info_t:

```
typedef struct ch_info_t
{
   int ch;
   int pos_x, pos_y;
   int dir; // Whether character moves vertically(1) or horizontally(0)
   int score;
   time_t last_fire_time;
   time_t stunned;
   int GAME_TOKEN;
} ch_info_t;
```

In the client only the ch variable is used so it knows what character is being controlled. All the rest is saved in the server. For the remaining replies, the server only sends a validation variable to confirm the message and processing was successful.

In the publisher/subscriber sockets the objects used are of type :

```
typedef struct screen_update_t
{
   int pos_x;
   int pos_y;
   char ch;
   char players[MAX_PLAYERS];
   int scores[MAX_PLAYERS];
   int player_count;
} screen_update_t;
```

They contain the information about every singular screen update, the type of update is encoded in the ch variable, as it can take the character to be printed in case of movement, zap or alien movement; ‘s’ in case oof a score update and ‘o’ in case of a game-over message.

5.3 Error treatment

In this section students should present what type of error treatments were implemented related to communication: validation of read data, verification of read/write return errors,
We validate all the returns of the send and receive functions across the program to make sure there are no communication errors. We do not however check the type of the received object, but we do handle faulty communications, for example we always check the message type when the server receives a message.

We also always check the returns of all the ZMQ functions such as socket and context creation.

...

6 PART B Communication

6.1 Sockets

In this section student should decribe any new socket in Part B.

New sockets utilized in part B

We create a new publisher socket in the server application to communicate with space-high-scores.C, space-high-scores.C has a subscriber socket to get the score updates as well as the game over messages.

6.2 Transferred data

In this section students should present the modified messages from part A to Part B.

Modified messages in part B

Between server and space-high-scores the communication is done via protocol buffers

```
message ScoreUpdate {
   repeated int32 scores = 1;
   repeated string characters = 2; }
```

The server is coded in C and the space-high-scores is coded in C++, so the communication requires a higher level of structure. This structure is also able to encode the game over messages by setting the number scores being bigger than 8, which was the approach used.

There is also the new message type for the case when the server decide to end the game.

6.3 Error treatment

In this section students should present what type of error treatments were implemented related to communication: validation of read data, verification of read/write return errors,

We maintain the processes used in part A, checking all the functions, all the new sockets and thread creation functions are also verified and dealt with accordingly in case of failure.

7 Conclusion

Final remarks about the project, what was learned, accomplishments, major difficulties and adversities.

This project allowed us to use various technologies learned in other modules in order to code a fully functioning system to play a game like space invaders. We were able to experiment with the versatility of different components such as the fork function and threads, and discover the advantages of using each one.

Another important point was how easy and flexible implementing inter language communication was, this will be useful for the future as different parts of other projects might be best suited for different programming languages.

The use of ZMQ also facilitates the use of sockets, whereas before the code using normal c sockets was much more verbose.

Finally, we were easily able to experience that most times, we can reutilize previous code, even when changing major characteristics of the program, and how vital it is to make sure adding code does not interfere with the previously developed programs.